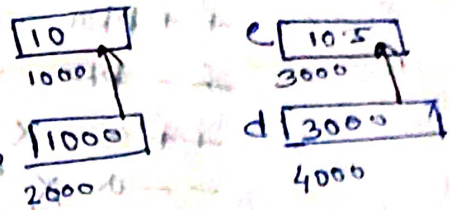
Note: in which the data should be stored.

Eg:
```
int a;
int *b;
float c = 10.5;
float *d;
b = &a;
d = &c;
```



(*) address will take 4 bytes. (in program we assume address 4 byte).

pf(a) = ? 10
→ sizeof(a) = 2B // size of a int 2 B
→ sizeof(c) = 4B // size of a float = 4B
→ size of (b) = 4B
→ size of (d) = 4B
→ size of (&a) = 4B
→ size of (&c) = 4B
→ size of (&d) = 4B.

■ Pointer to array :- → Array is collection of homogeneous elements.

→ Mem for array elements is continuously.
will be allocated

① Array name w/o subscript contains base address of the array allocated by the compiler.
② This base addr is a constant and it can't be changed.
③

```
void main(){
    int a[5] = {10, 20, 30, 40, 50};
    int *ptr = &a[0];
    ++ptr;
    ++*ptr;
    *ptr++;
    pf("%d", *ptr);
}
```

```
int *ptr;
ptr = &a[0];
```
↕
```
int *ptr;
ptr = a;
```

associativity of unary operator right to left.

o/p: 30 .29

pf(a) = 1000
pf(ptr) = 1000

Note: ① When the pointer variable is ↑ then it will ↑ w.r.t. size of the elements to which it is pointing to.
② Unary operator associativity is R→L.

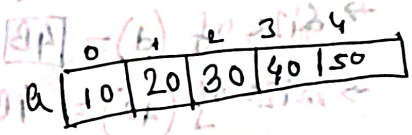
③ For the above operations, ptr is a pointer variable but a is a pointer constant

++ptr ✓
--ptr ✓
++a ✗
--a ✗

④ Array name is given as constant pointer, because i.e the only source in the mem how the arrays contents can be accessed.

```
Void main( ) {
  int a[5] = {10, 20, 30, 40, 50};
  printf("%d %d %d %d", a[2], 2[a], *(a+2), *(2+a));
}
```

a[2]	a[2]
*(a+2)	*(a+2)
*(1000+2)	*(2+a)
*(1004)	2[a]
30	

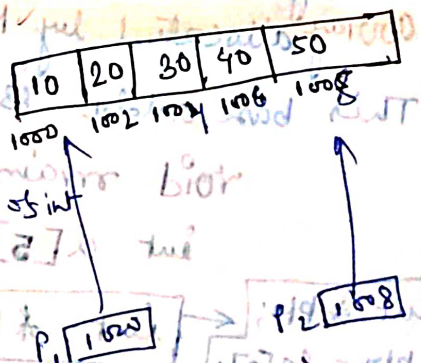


O/P: 30, 30, 30, 30

Pointer Arithmetic:-

```
Void main( ) {
  int a[5] = {10, 20, 30, 40, 50};
  int *p1 = a;
  int *p2 = a + 4;
}
```

1000
1008
(a[4])
1000 + 4*2 → sizeof int
= 1008



① We can add integer constant to a pointer, it means skipping the element in a forward direction.

Eg: p1 + 2
1000 + 2 = 1004
p1

② We can substract integer constant from pointer it means skipping the elements in backward direction.

1008 - 2

1008 − 2 * 2

= 1008 − 4

= 1004

③ We can subtract two pointers, this is allowed only when both are pointing to same array & $P_2 \geq P_1$.

This operation will give us # elements b/w from P_2 before P_1

$P_2 - P_1$

$1008 - 1000 = 8 = \frac{8}{2} = 4$ ✓

$P_2 \geq P_1$

$1008 - 1008 = 0 = \frac{0}{2} = 0$

④ $P_1 + 2.5$ ✗ We can't add floating point number to the pointer becoz it has no meaning.

⑤ We can't perform +, *, / b/w two pointers becoz it has no meaning.

$P_1 + P_2$ ⊗

$P_1 * P_2$ ⊗

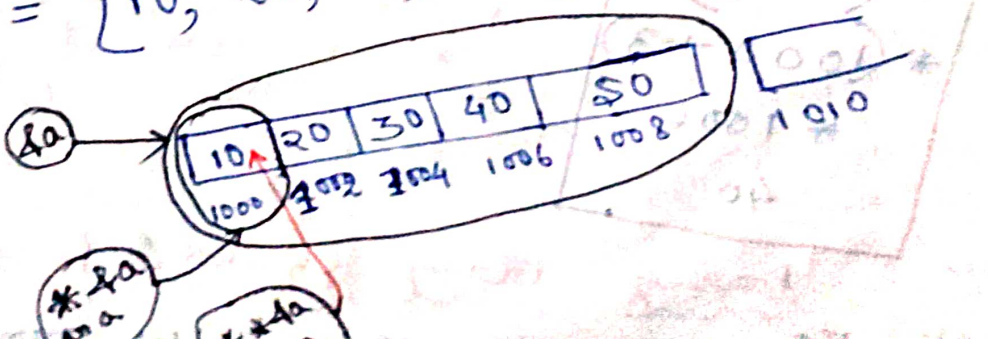
P_1 / P_2 ⊗

Eg:
```
void main(){
    int a[5] = {10,20,30,40,50};
    int *p1 = a;
    int *p2 = a+4;
    printf("%d", p2-p1);
}
```

O/P: 1008 − 1000 = 8 = 8/2 = 4.

One dimentional Array

int a[5] = {10, 20, 30, 40, 50}



Expression	print value	Type	Meaning
1. &a	1000	int *[] (pointing entire one dimensional array)	Base addr of entire one dimensional array.
2. *&a	1000	int *	Base addr of the element of the array
3. a	1000	int *	Base addr of the 0th element of the array
4. *a	10	int	Value of the 0th element of the array
5. &a+1	1010	int *[]	Base addr of the next contiguous 1D array.
6. a+1	1002	int *	Base addr of the 1st element of the array incrementing the value of 0th element.
7. *a+1	11	int	
8. **a	error/ garbage value	?	outside the array mem
9. a+3	1006	int *	base addr of the 3rd ele of the array
10. *(a+3)	40	int	Value of the 3rd element of array.
11. a[3]	40	int	Value of the 3rd element of the array.

*(a+3)
*(1000+3)
*1006
40

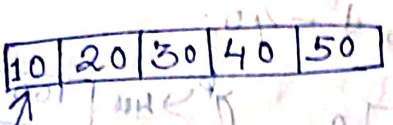
Lec-9 Passing array pointer to function:

Note: If u want to pass all the array elements to a function, then it is not a good idea to pass all the array

elements, instead by simply passing base address of the array we can access all the array elements.

Eg:
```
void main(){
    int a[5] = {10,20,30,40,50};
    display(a,5);
}
void display(int *p, int n){
    int i;
    for(i=0; i<n; i++){
    printf("%d", p[i]);
}
```

What is the o/p printed?



p → 1100

Gate – 20

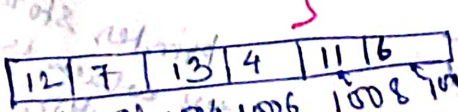
10
20
30
40
50

```
void main(){
    int a[6] = {12,7,13,4,11,6};
    printf("%d", f(a,6));
}
```

What is the o/p printed?

```
int f(int *a, int n){
    if(n<=0)
        return 0;
    else if(*a%2==0)
        return *a + f(a+1, n-1);
    else
        return *a - f(a+1, n-1);
}
```

15



f(a,6) = 12 + f(1002,5) = 15
7 - f(1004,4) = 3
6 + f(1012,0) = 4
13 - f(1006,3) = 4

Array of pointers:

Just like we had a normal integer array, similarly we have array of pointers.

Eg: int a[5]

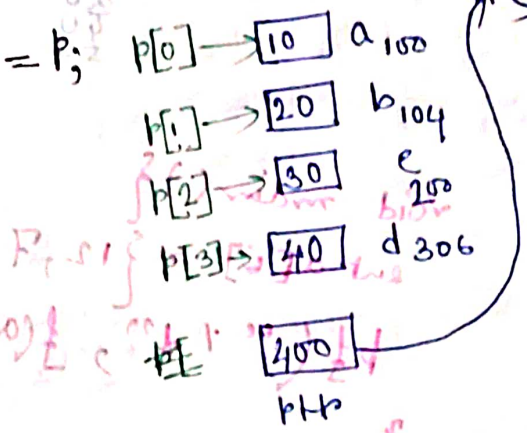
int *a[5].

- Between operators & operand operand has high precedence.
- Between * and Braces '[ ]' has high precedence.

[Download precedence of operators] and print it.

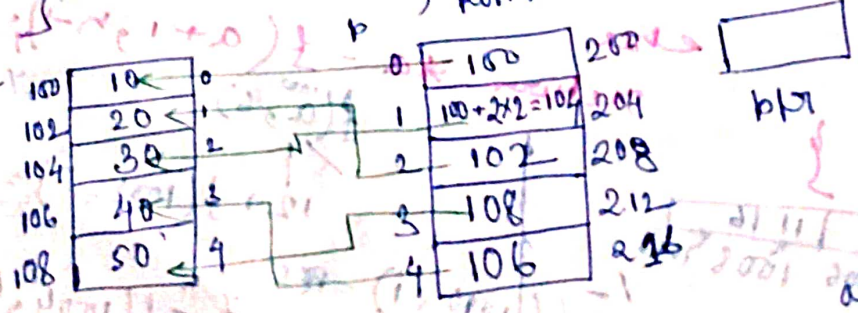
- 'a' is a array of pointers of type integer, where we can store addrs of 5 integers.

Eg:
```
main(){
①   int a=10, b=20, c=30, d=40;
    int *p[4];
    int **ptr = p;
    p[0] = &a;
    p[1] = &b;
    p[2] = &c;
    p[3] = &d;
}
```



Eg②
```
main(){
    int a[5] = {10, 20, 30, 40, 50};
    int *p[5] = {a, a+2, a+1, a+4, a+3};
    int **ptr = p;
}
```

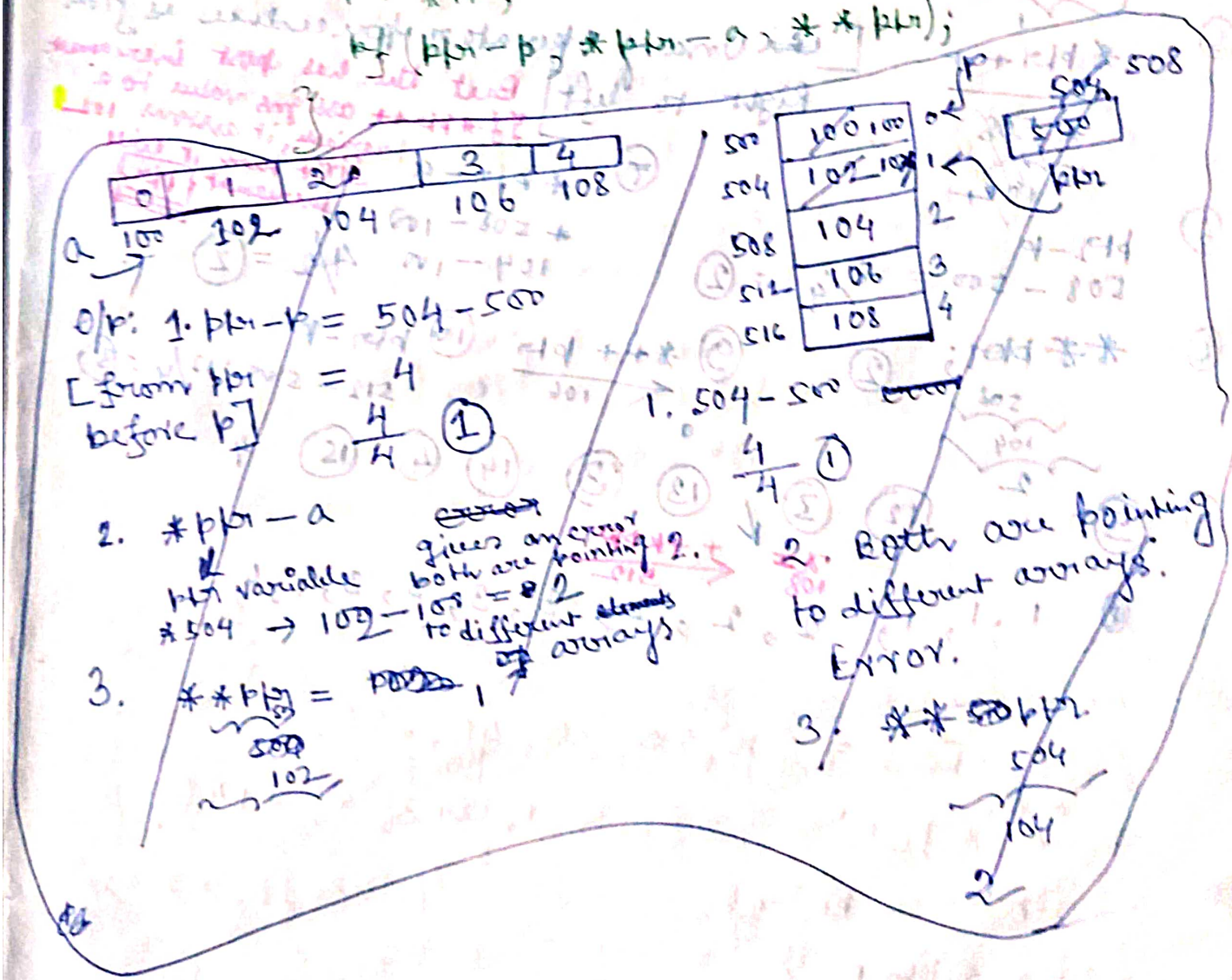
→ pointer which point an integer pointer.

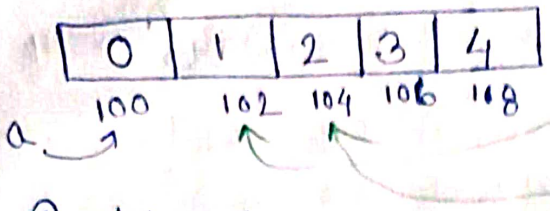


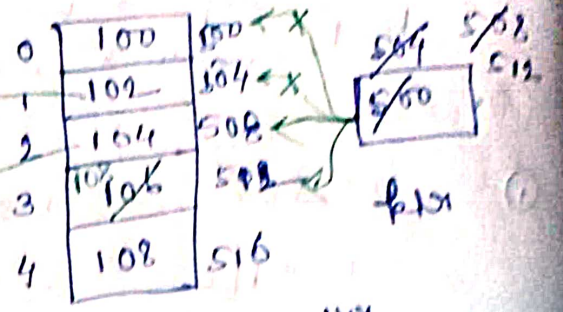
pointer stores addrs which is of 4 bytes.

```
void main ()
{
    int a[5] = {0, 1, 2, 3, 4};
    int *p[5] = {a, a+1, a+2, a+3, a+4};
    int **ptr = p;

    ptr++;
    printf("%d %d", ptr - p, **ptr);
    *ptr++;
    printf("%d %d", ptr - p, **ptr);
    *++ptr;
    printf("%d %d", ptr - p, **ptr);
    ++*ptr;
    printf("%d %d", ptr - p, **ptr);
```







① ptr++
500

② 504 − 500 = 4
4/4 = ①

③ *ptr − a
504
102 − 100
2/2 = ①

[Unary operator has higher precedence]
[Both are pointing to same array, so subtraction is allowed]
[(from *ptr to a) how many elements the array contains]

④ **ptr ①
504
102
1

⑤ *ptr++
[Unary operator precedence is from Right to left] But this has post increment. If *ptr++ assigns value to a variable, it assigns 102 first then it will increment ptr.

⑥ ptr2 − p
508 − 500 = 8/4 = ②

⑦ *ptr − a
*508 − 100
104 − 100 4/2 = ②

⑧ **ptr; ②
508
104
2

⑨ *++ptr
106

⑩ ptr − p
512 − 500 ÷ 12/4 = ③

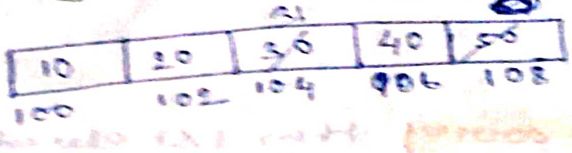
⑪ ③ ⑫ ③ ⑬ ③ ⑭ ④ ⑮
108
++*ptr
512

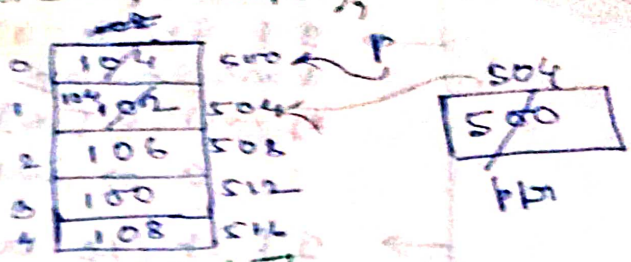
⑯ 1, 1, 1, 2, 2, 2, 3, 3, 3, 3, 4, 4

⑰
```
main(){
int a[5] = {10, 20, 30, 40, 50};
int *p[5] = {a+2, a+1, a+3, a, a+4};
int **ptr = p;
**ptr++;
printf("%d, %d, %d", ptr-p, *ptr-a, **ptr);
*++*ptr;
printf("%d, %d, %d", ptr-p, *ptr-a, **ptr);
```

→ → * * ptr;

printf (ptr - p, * ptr - a, * * ptr);





[Unary operators precedence Right to left]

But this is post increment, after executing the line the ptr will be incremented.

if x = * * ptr + + ;

① 1, 1, 20

② 1, 29, 30

③ 1, 29, 31

---

⑨ main ()

{

int a[5] = { 50, 20, 36, 80, 75 };

int * p[5] = { a+4, a+3, a+1, a+2, a };

int * * ptr = p;

* + + ptr;

if (ptr - p, * ptr - a, * * ptr);

* + + ptr;

if (ptr - p, * ptr - a, * * ptr);

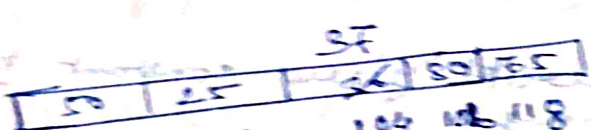
+ + * ptr;

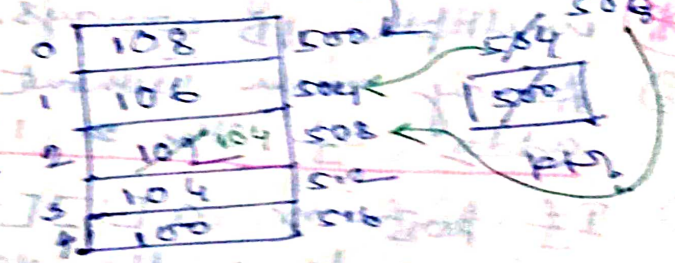
if (ptr - p, * ptr - a, * * ptr);

+ + * * ptr;

if (ptr - p, * ptr - a, * * ptr);

}





① 1, 3, 80

② 2, 1, 25

③ 2, 2, 36

④ 2, 2, 50

Multi dimensional Array

Array name w/o subscript always gives base addr of first element

First element

→ If it is 1D array, then 1st element means base addr of 0th element.

Eg: int a[5];

a ⇒ int *

→ If it is a 2D array then 1st element is base addr of 0th 1D array.

Eg: int a[3][4];

a ⇒ int *[4];

→ If it is 3D array, then first element means base addr of 0th 2D array.

Eg: int a[3][4][5];

a ⇒ int *[][];

2D array :→ Eg: int a[3][3] = {10, 20, ..., 90}

NOTE: ① A good example for the two dimensional array is pointer.

② The above example represents 3 rows & 3 columns.

③ In the 2d array a[i] represent constant base addr of ith row allocated by the compiler & it can't be changed.

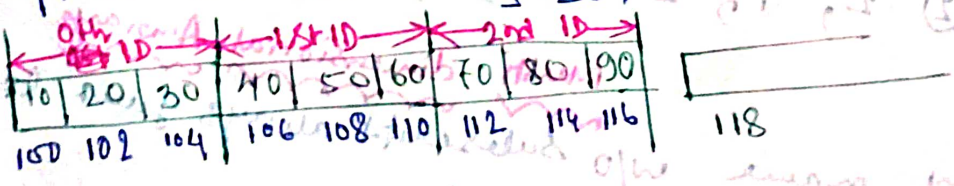
④ If we perform a[i] = 2000; This gives an error. Bcz we can't change the *, constant base addr of ith row.

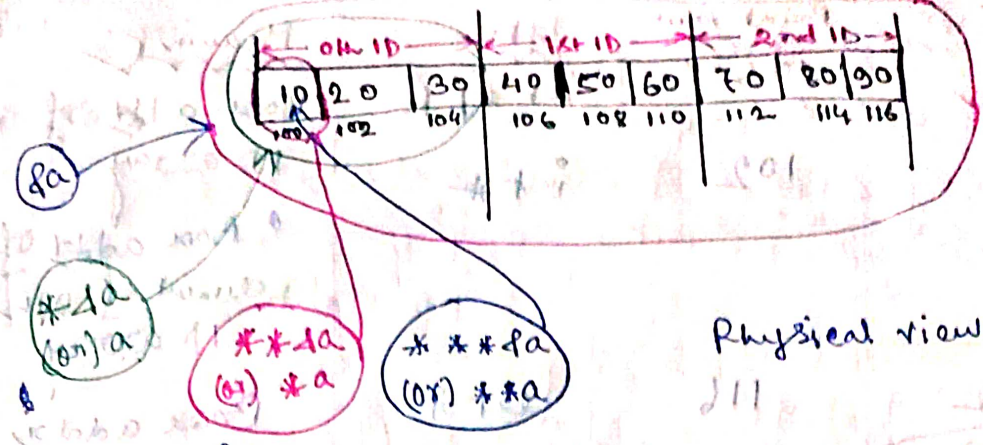
⑤ The 2D array stored in the memory in the form of

(*) multiple 1D arrays.

⑥ Array name without subscript gives constant base addr of 0th 1D array.

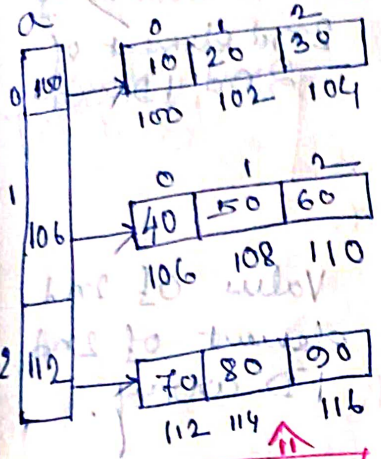
⑦ If perform a[i][j] = 20; // ith, jth columns will replace the value of 20.





147.

Physical view.



Logical view

Dt: 15.05.21

~~Expression~~ ~~Point value~~ ~~Type~~ ~~Meaning~~

1) ~~&a~~ ~~100~~ ~~int[][]~~

	Expression	Point value	Type	Meaning
1)	&a	100	int *[][]	Base addr of entire 2D array.
2)	*&a	100	int *[]	Base addr of 0th 1D array.
3)	a	100	int *[]	Base addr of 0th 1D array.
4)	*a	100	int *	Base addr of 0th element of 0th 1D array.
5)	**a	10	int	value of 0th element of 0th 1D array
6)	a[0] ≡ *(a+0) ≡ *a	100	int *	Base addr of 0th element of 0th 1D array.
7)	&a+1	118	int *[][]	Next continuous 2D array.

Expression	Print value	Type	Meaning
(8) a+1	106	int *[]	Base addr of 1st 1D array
(9) *a+1	102	int *	Base addr of 1st element of 0th 1D array
(10) *(a+2)+2 = *(a+2*6)+2 = *112+2 =	116	int *	Base addr of 2nd element of 2nd 1D array.
(11) *(*(a+2)+2)	90	int	Value of 2nd element of 2nd 1D array.

a[2][2]

((a+2)+2)

((100+2)+2)

⇓

((112)+2)

*(112+2)

*(116)

90

[One bracket means one pointer]

Eg: a[1][1]

((a+1)+1) → pointing to the entire 0th 1D array

((100+6)+1)

*(*106+1) → pointing to the entire 1st 1D array

*(106+1) → " " " 0th element of 1st 1D array

*(108) → " " " 0th element element of 1st 1D array

50 → value of the 1st element of 1st 1D array.

Expr		Type	Meaning
12) *(a+1)+2	110	int *	B.A of 2nd element of 1st 1D array
13) ***a	error (most of the time) local compiler G.V ← (some time)	?	Referring garbage location or pointing outside array memory

GATE

①

```
void main(){
    int a[5][3] = {10, 20, 30, ... 150};
    pf("%d", ((a == a[0] && (a[0] == *a)));
```

What is the output printed?

ⓐ 1
ⓑ 0
ⓒ compilation Error
ⓓ Run time error

During comparison, the point value will be compared. The type isn't mandatory here. The type is mandatory when + opⁿ is done. Eg: (a+1).

$a[0] = *(a+0)$
$*a$

ⓐ 1

GATE 2003

②

int *A[10]; → pointer array
int B[10][10]; → 2D array

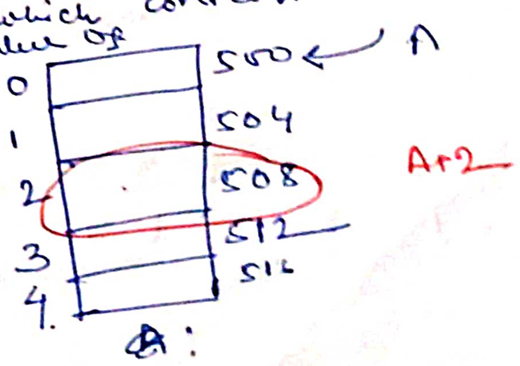
of the following expressions —

Ⓘ A[2] Ⓘⓘ A[2][3] Ⓘⓘⓘ B[1] Ⓘⓥ B[2][3]

Which of the following expression will not give compile time error, if we use them as left side of assignment statement in C program?

~~A[10]~~ int * A[10] → A is a pointer Array which contains 10 int *.

Ⓘ ✓ $A[2] = *[A+2]$ ~~pointing to~~ value of 2nd element. 0



Ⓘⓘ ✗ $A[2][3] = *[*[A+2]+3]$
$*[500+2]$
$*508$
G.V. + 3 → error

Ⓘⓘⓘ ✓ $B[1] = *(B+1)$ → pointing to 0th element of 1st 1D array.

Ⓘⓥ ✓ $B[2][3] = *(*(B+2)+3)$ → ~~pointing to~~ value of 3rd element of 2nd 1D array.

~~Ⓘⓘ & Ⓘⓘⓘ will give compile time error.~~ Ⓘ & Ⓘⓥ will not give compile time error.

int * A[10];

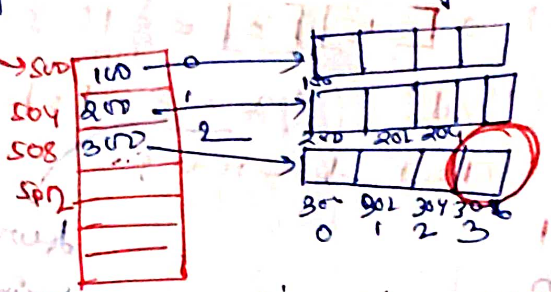
declaration-এ থাকলে → left to right যাবো, Operand মেলবো যেখানে,

→ Operand having higher precedence than Operator.

→ A-র দুদিকের operator আছে, [ ] has higher precedence

→ A is an array of pointers of type integer.

(II) A[2][3]



compile time error নেই।

It will not give any compile time error.

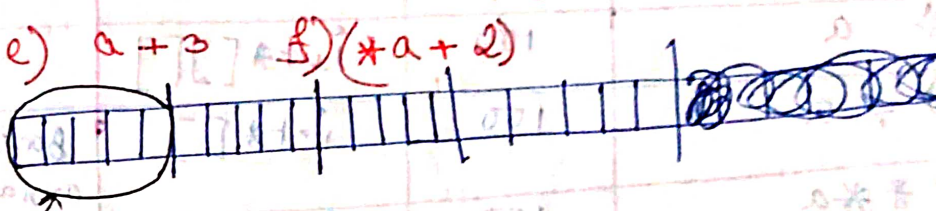
H.W : ① int a[4][5] = {1, 2, 3 . . . 20};

print all expressions, B.A = 100, size of elements = 2B.

② int a[4][5] = {1, 2, 3, ... 20}; B.A = 100, size of elements = 2B.

(a) *(a+2)+1 (b) *a+3 (c) *(*(a+3)+4)

(d) *(a+1)+3. e) a+3 f) (*a+2)



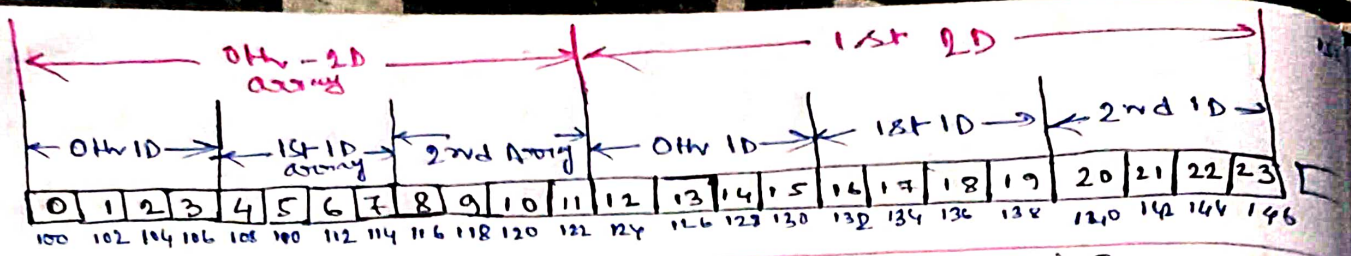
(a) *(a+2)+1 pointing to the 1st element of 2nd 1d array.

3D Array

① Array name without subscript B.A of 0th 2D array

② 3D array will be stored in the memory in the form of multiple 2D arrays.

Eg: int a[2][3][4]



	Expression	R.V.	Type	Meaning
1)	&a	100	int *[][][]	Base Addr of entire 3D array.
2)	*&a	100	int *[][]	Base addr of 0th 2D array.
3)	a	100	int*[][]	99
4)	*a	100	int*[]	Base addr of 0th 1D array of 0th 2D array
5)	**a	100	int*	B.A of 0th element of 0th 1D array of 0th 2D array
6)	***a	0	int	value of 0th element of 0th 1D array of 0th 2D array.
7)	****a	error	?	referring unknown location or outside array mem
8)	&a+1	~~147~~ 148	int *[][][]	Base addr of next 3D array. (continuous)
9)	a+1	124	int *[][]	Base addr of ~~next~~ 1st 2D array.
10)	*a+1	108	int *[]	Base Addr of 1st 1D array of 0th 2D array.
11)	**a+1	102	int*	Base Address of 1st element of 0th 1D Array of 0th 2D array
12)	*(a+1)+2	140	int *[]	B.A of ~~0th element of~~ 2nd 1D array of 1st 2D array.
13)	*(*(a+1)+2)	40	int *	B.A. of 0th element of 2nd 1D array of 1st 2D array.
14)	*(*(*(a+1)+1)+1)	18	int	value of 2nd element of 1st 1D array of 1st 2D array